Corrigé TP4

1. Le voici :

```
let a1 =
{nb_lettres = 3; nb_etats = 3; etat_initial = 0;
 etats_finaux = [|false; false; true|];
 delta = [|[|2; 0; -1|]; [|0; -1; 2|]; [|2; -1; 1|]|]}
```

2. En suivant les instructions de l'énoncé, on remarque que le seul état final de A_2 est inaccessible, d'où on déduit que cet automate ne reconnaît aucun mot.

3. Il suffit de lire le mot lettre à lettre en s'arrêtant dès qu'un blocage survient :

```
let rec delta_star (a:automate_det) (q:etat) (m:mot) :etat option =
  match m with
  |[] -> Some q
  |l::m' -> let etat_suivant = a.delta.(q).(l) in
            if etat_suivant = -1 then None
            else delta_star a etat_suivant m'
```

4. Il suffit de vérifier si la lecture du mot conduit à un état final à l'aide de delta_star :

```
let accepte (a:automate_det) (m:mot) :bool =
  match (delta_star a a.etat_initial m) with
  |None -> false
  |Some q -> a.etats_finaux.(q)
```

5. Un parcours en profondeur à partir de l'état initial permet de répondre à la question. Il faut être capable de réimplémenter un tel parcours en 10 minutes.

```
let etats_accessibles (a:automate_det) :bool array =
  let vus = Array.make a.nb_etats false in
  let rec parcours_profondeur (q:etat) =
    if q != -1 && not vus.(q) then
      begin
        vus.(q) <- true;
        Array.iter parcours_profondeur a.delta.(q)
      end
  in
  parcours_profondeur a.etat_initial;
  vus
```

   On rappelle au passage le fonctionnement de Array.iter : cette fonction permet d'appliquer un même traitement à tous les éléments d'un tableau. Dans notre cas, on applique un parcours en profondeur à tous les voisins du sommet q : ce sont précisément ceux stockés dans la ligne q dans delta.

6. Le langage reconnu par un automate déterministe est vide si et seulement si aucun état final n'est accessible depuis l'état initial. D'où la proposition suivante :

```
let langage_non_vide (a:automate_det) :bool =
  let accessibles = etats_accessibles a and res = ref false in
  for i = 0 to a.nb_etats -1 do
    res := !res || (accessibles.(i) && a.etats_finaux.(i))
  done;
!res
```

On peut aussi exploiter les fonctions du module Array : Array.mapi permet d'appliquer une fonction à tous les éléments d'un tableau à partir de cet élément et de son indice (d'où le "i") et Array.mem teste l'appartenance d'un élément à un tableau :

```
let langage_non_vide (a:automate_det) :bool =
  Array.mem true (Array.mapi (fun i q -> a.etats_finaux.(i) && q)
                 (etats_accessibles a))
```

7. On constate qu'effectivement langage_non_vide a2 renvoie true, ce à quoi on s'attendait depuis la question 2. La complexité de cette fonction est en $O(mn)$ où n est le nombre d'états de l'automate en entrée et m le nombre de lettres de l'alphabet sur lequel il est défini. En effet, la complexité de cette fonction est majorée par celle du parcours en profondeur effectué par etats_accessibles et ce dernier traite au plus une fois les nm transitions de l'automate, chacune en temps constant.

8. Laissé en exercice. On conçoit une fonction etats_coaccessibles qui transpose le graphe sous-jacent à l'automate en entrée et, pour chacun des états finaux, effectue un parcours en profondeur depuis cet état. Les états visités lors de ces parcours sont les états coaccessibles. Il ne reste plus qu'à vérifier que tous les états sont à la fois accessibles et coaccessibles pour vérifier l'émondage.

9. Il suffit de rendre finaux tous les états qui ne l'étaient pas et de supprimer le caractère final de ceux qui l'étaient. Afin que les transitions de l'automate complémentaire et de l'automate en entrée soient complètement décorrélées, on recopie ces dernières.

```
let complementaire (a:automate_det) =
  let n = a.nb_etats and m = a.nb_lettres in
  let complementaire_finaux = Array.map not a.etats_finaux
  and transitions = Array.init n (fun i -> Array.copy a.delta.(i))
  in {nb_lettres = m; nb_etats = n; etat_initial = a.etat_initial;
      etats_finaux = complementaire_finaux; delta = transitions}
```

10. On construit l'automate produit comme vu en cours, avec une facilité supplémentaire dans la confection de ses transitions qui est qu'on n'aura jamais de blocage, les automates en entrée étant complets. La fonction (bijective) traduction permet de renommer les couples d'états de a1 et a2 de sorte à ce qu'ils soient bien des entiers entre 0 et le nombre d'états de l'automate produit moins un.

    Un état est final dans l'automate produit s'il est final dans les deux automates, d'où le calcul de finaux_inter. Les transitions sont obtenues en lisant simultanément une même lettre depuis deux états : un dans chaque automate. Cette remarque permet de construire transitions_inter.

```
let intersection (a1:automate_det) (a2:automate_det) :automate_det =
  let m = a1.nb_lettres and n = a1.nb_etats * a2.nb_etats in
  let traduction i j = i*a2.nb_etats + j in
  let initial_inter = traduction a1.etat_initial a2.etat_initial in
  let finaux_inter = Array.make n false in
```

```
  for i = 0 to a1.nb_etats -1 do
    for j = 0 to a2.nb_etats -1 do
      finaux_inter.(traduction i j) <- a1.etats_finaux.(i) && a2.etats_finaux.(j)
    done;
  done;
  let transitions_inter = Array.make_matrix n n (-1) in
  for i = 0 to a1.nb_etats -1 do
    for j = 0 to a2.nb_etats -1 do
      for l = 0 to m-1 do
        let etat_origine = traduction i j in
        let etat_destination = traduction a1.delta.(i).(l) a2.delta.(j).(l) in
        transitions_inter.(etat_origine).(l) <- etat_destination
      done;
    done;
  done;
  {nb_lettres = m; nb_etats = n; etat_initial = initial_inter;
   etats_finaux = finaux_inter; delta = transitions_inter}
```

11. Si L et L' sont deux langages sur Σ on a :

$$L \subset L' \text{ si et seulement si } L'' = L \cap (L')^c = \emptyset$$

Si $L \subset L'$, supposons par l'absurde que $L \cap (L')^c \neq \emptyset$. Alors il existe x tel que $x \in L$ et $x \notin L'$ ce qui contredit $L \subset L'$. Réciproquement, si $L \cap (L')^c = \emptyset$ et que $x \in L$ alors $x \notin (L')^c$ donc $x \in L'$.

12. Il suffit d'utiliser la caractérisation obtenue ci-dessus et d'exploiter les fonctions complementaire, intersection et langage_non_vide précédemment implémentées :

```
let inclusion_langages (a1:automate_det) (a2:automate_det) =
  let a = intersection a1 (complementaire a2) in not (langage_non_vide a)
```

13. Un automate est complet si et seulement si il n'y a aucun blocage, c'est-à-dire si et seulement si il n'y a aucun -1 dans la table de ses transitions vu nos conventions :

```
let est_complet (a:automate_det) :bool=
  let res = ref true in
  for q = 0 to a.nb_etats -1 do
    for l = 0 to a.nb_lettres -1 do
      res := !res && (a.delta.(q).(l) != (-1));
    done;
  done;
!res
```

On peut aussi exploiter Array.for_all en cascade pour vérifier que toutes les cases de la matrice delta sont différentes de -1 (ou un combinaison de Array.exists) :

```
let est_complet (a:automate_det) :bool =
  Array.for_all (fun tab -> Array.for_all (fun i -> i != (-1)) tab) a.delta
```

14. Pour compléter, on ajoute un état puits à l'automate initial s'il n'était pas déjà complet, ce qui revient à ajouter un état étiqueté par n si l'automate initial avait n états. On recopie ensuite les informations relatives aux états finaux puis toutes les transitions en redirigeant les blocages sur ce nouvel état.

Remarquons qu'appliquer ce traitement directement à l'automate même s'il est complet produit bien un automate complet : on pourrait donc se passer du premier test.

```
let complete (a:automate_det) :automate_det =
  if est_complet a then a else
    let n = a.nb_etats and m = a.nb_lettres in
    let nouveaux_finaux = Array.make (n+1) false in
    for i = 0 to n-1 do
      nouveaux_finaux.(i) <- a.etats_finaux.(i)
    done;
    let nouvelles_transitions = Array.make_matrix (n+1) m n in
    (*par défaut, on va vers l'état puits pour initialiser correctement la dernière ligne*)
    for q = 0 to n-1 do
      for l = 0 to m-1 do
        let q' = a.delta.(q).(l) in
        if q' = -1 then nouvelles_transitions.(q).(l) <- n
        else nouvelles_transitions.(q).(l) <- q'
      done;
    done;
    {nb_lettres = m; nb_etats = n+1; etat_initial = a.etat_initial;
     etats_finaux = nouveaux_finaux; delta = nouvelles_transitions}
```

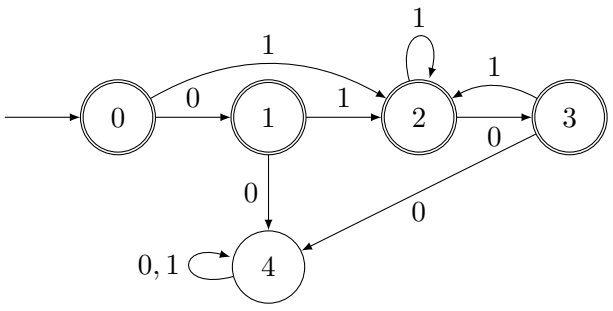
15. Tester l'égalité de deux langages revient à tester une double inclusion ce que inclusion_langages permet justement de faire. Cette fonction n'acceptant que des automates complets en entrée, on commence par compléter les automates dont on veut tester l'équivalence :

```
let egalite_langages (a1:automate_det) (a2:automate_det) =
  let a1_complet = complete a1 and a2_complet = complete a2 in
  (inclusion_langages a1_complet a2_complet)
  && (inclusion_langages a2_complet a1_complet)
```

Si A est un automate à n états et m lettres, le complémentariser se fait en $O(mn)$ (à cause de la recopie des transitions), le compléter aussi, et déterminer si le langage qu'il reconnait est vide aussi d'après la question 7. De plus, intersecter un automate à n_1 états et un automate à n_2 états sur un alphabet à m lettres se fait en $O(n_1 n_2 m)$ (là encore, le calcul des transitions majore le temps de calcul) et produit un automate à $n_1 n_2$ états.

Ainsi, tester l'équivalence de A_1 et A_2 se fait en $O(n_1 n_2 m)$ où n_1 (resp. n_2) est le nombre d'états de A_1 (resp. A_2) et m est le nombre de lettres de l'alphabet commun sur lequel ils sont définis.

16. Après déterminisation on obtient l'automate déterministe et complet A_1' suivant :



17. Les automates A_1' et A_1 sont équivalents. De plus A_1' et A_2 sont déterministes. On peut donc appliquer egalite_langages à A_1' et A_2 et le résultat obtenu (true) nous indique que A_1 et A_2 sont bien équivalents. La même méthode montre que A_1 et A_3 sont équivalents.

18. Ces trois automates reconnaissent le même langage d'après la question précédente à savoir celui reconnu par A_3 dont il n'est pas difficile de constater qu'il correspond au langage sur $\{0, 1\}$ des mots qui ne contiennent pas le facteur 00.